COMPLETE COURSE NOTES

SOFTWARE ARCHITECTURE & DESIGN

SDLC, UML, Patterns & Best Practices

"Designing Robust Systems for the Modern World"

First Edition - 2026

Written & Compiled By

Gohar Rehman

Software Engineering Student

Government Graduate College Samnabad, Faisalabad

Software Architecture & Design

Complete Course Notes

Author: Gohar Rehman

Program: Software Engineering

Institution: Government Graduate College Samnabad, Faisalabad

Edition: First Edition | **Year:** 2026

Table of Contents

Module	Topic
Module 0	**Introduction to SDLC**
1	Introduction to SDLC & Software Engineering
Module 1	**Foundations of Design & Architecture**
2	Foundations of Design & Architecture
Module 2	**Modeling & Object-Oriented Design**
3	Modeling & Object-Oriented Design (OOD)
Module 3	**Architectural Styles & Patterns**
4	Architectural Styles, Patterns & Qualities
Module 4	**Specialized Design Layers**
5	Specialized Design Layers
Module 5	**Process, Implementation & Evaluation**
6	Process, Implementation & Evaluation
	References

INTRODUCTION TO SDLC & SOFTWARE ENGINEERING

1. Software Engineering Overview

1.1 Definition and Scope

Software Engineering is defined by IEEE as "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software."

It is not just about writing code; it encompasses the entire lifecycle of a software product. It bridges the gap between unstructured programming and disciplined engineering.

Key Objectives:

- **Maintainability**: Software should be written in such a way that it can evolve to meet changing needs.
- **Dependability and Security**: Software should not cause physical or economic damage in the event of system failure.
- **Efficiency**: Software should not make wasteful use of system resources such as memory and processor cycles.
- **Acceptability**: Software must be acceptable to the type of users for which it is designed.

1.2 The Software Crisis

The term "Software Engineering" was coined to address the "Software Crisis" of the 1960s, characterized by:

- Projects running over budget.
- Projects running over time.
- Software being inefficient and of low quality.
- Software often not meeting requirements.

1.3 Software Myths

- **Management Myth**: "We have standards and procedures for building software, does that provide my people with everything they need to know?" (Reality: Standards are often ignored or outdated).
- **Customer Myth**: "A general statement of objectives is sufficient to begin writing programsâ€”we can fill in the details later." (Reality: Poor requirements are the #1 cause of failure).

- **Developer Myth**: "Once we write the program and get it to work, our job is done." (Reality: 60-80% of effort is spent after the first delivery).

---

2. Software Development Life Cycle (SDLC)

The **Software Development Life Cycle (SDLC)** is a framework defining tasks performed at each step in the software development process. It is a structure followed by a development team within the software organization.

2.1 Detailed Phases of SDLC

Phase 1: Requirement Analysis

This is the most important and fundamental stage. It involves gathering requirements from customers, market research, and analysis.

- **Activities**: Interviews, surveys, analyzing existing systems.
- **Key Question**: "What do we need to build?"
- **Output**: **SRS (Software Requirement Specification)** document.

Phase 2: Feasibility Study

Once requirements are gathered, the team analyzes if the project is feasible.

- **Technical Feasibility**: Do we have the technology/skills?
- **Operational Feasibility**: Will it work in the user's environment?
- **Economic Feasibility**: Is it cost-effective?
- **Output**: Feasibility Report.

Phase 3: Design

The requirements are translated into a blueprint. This phase defines the architecture, components, interfaces, and data.

- **High-Level Design (HLD)**: Architecture, database design, module list.
- **Low-Level Design (LLD)**: Logic of each module, algorithms, interface details.
- **Output**: **DDS (Design Document Specification)**.

Phase 4: Implementation (Coding)

Developers write the actual code based on the design documents.

- **Activities**: Programming, code review, unit testing.
- **Key Goal**: Produce working software.
- **Output**: Source Code.

Phase 5: Testing

The code is tested against the requirements to ensure it is bug-free.

- **Types**: Unit Testing, Integration Testing, System Testing, Acceptance Testing.
- **Output**: Test Reports, Bug Reports.

Phase 6: Deployment

The software is released to the production environment for users.

- **Strategies**: Direct cutover, Phased rollout, Pilot study.

Phase 7: Maintenance

This is the longest phase. It involves fixing bugs found after delivery and upgrading the system.

- **Corrective**: Fixing bugs.
- **Adaptive**: Modifying for new environments (e.g., new OS).
- **Perfective**: Improving performance or adding features.

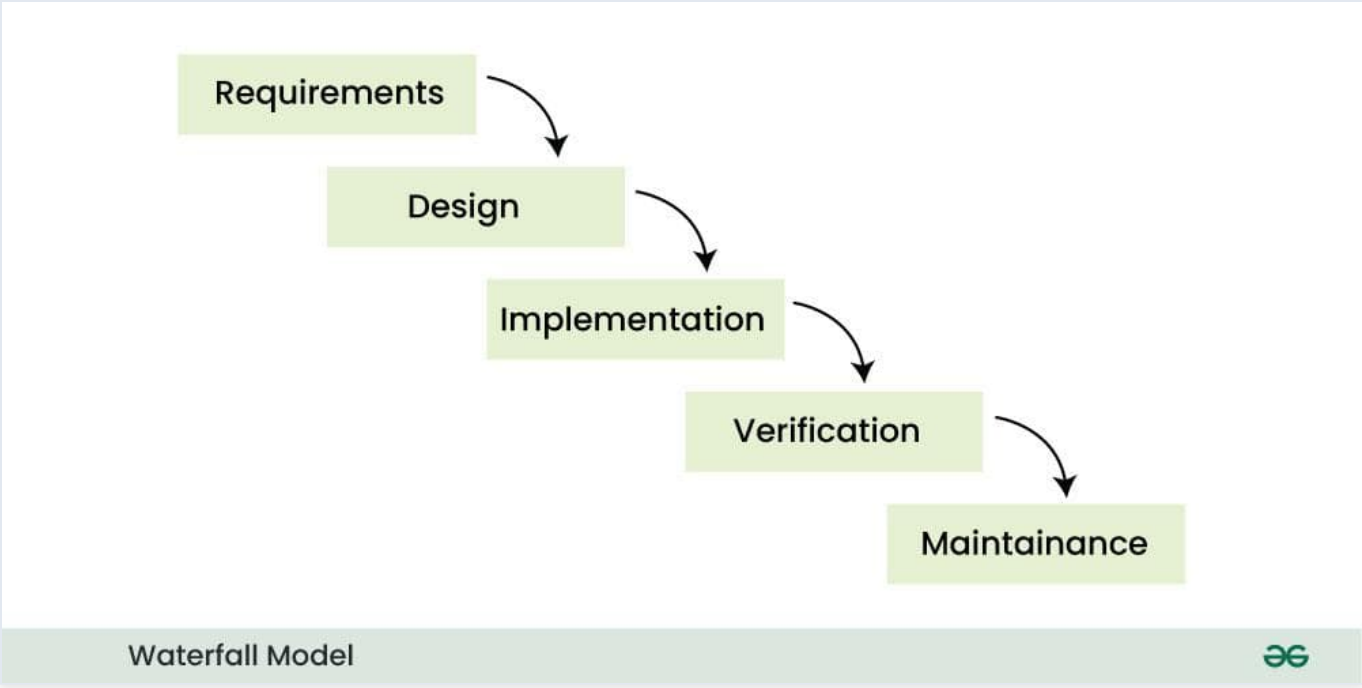
```
graph TD
    A[Requirement Analysis] -->|SRS| B[Feasibility Study]
    B -->|Report| C[Design]
    C -->|DDS| D[Implementation]
    D -->|Code| E[Testing]
    E -->|Verified Code| F[Deployment]
    F --> G[Maintenance]
```

---

3. SDLC Models

3.1 Waterfall Model (Linear Sequential)

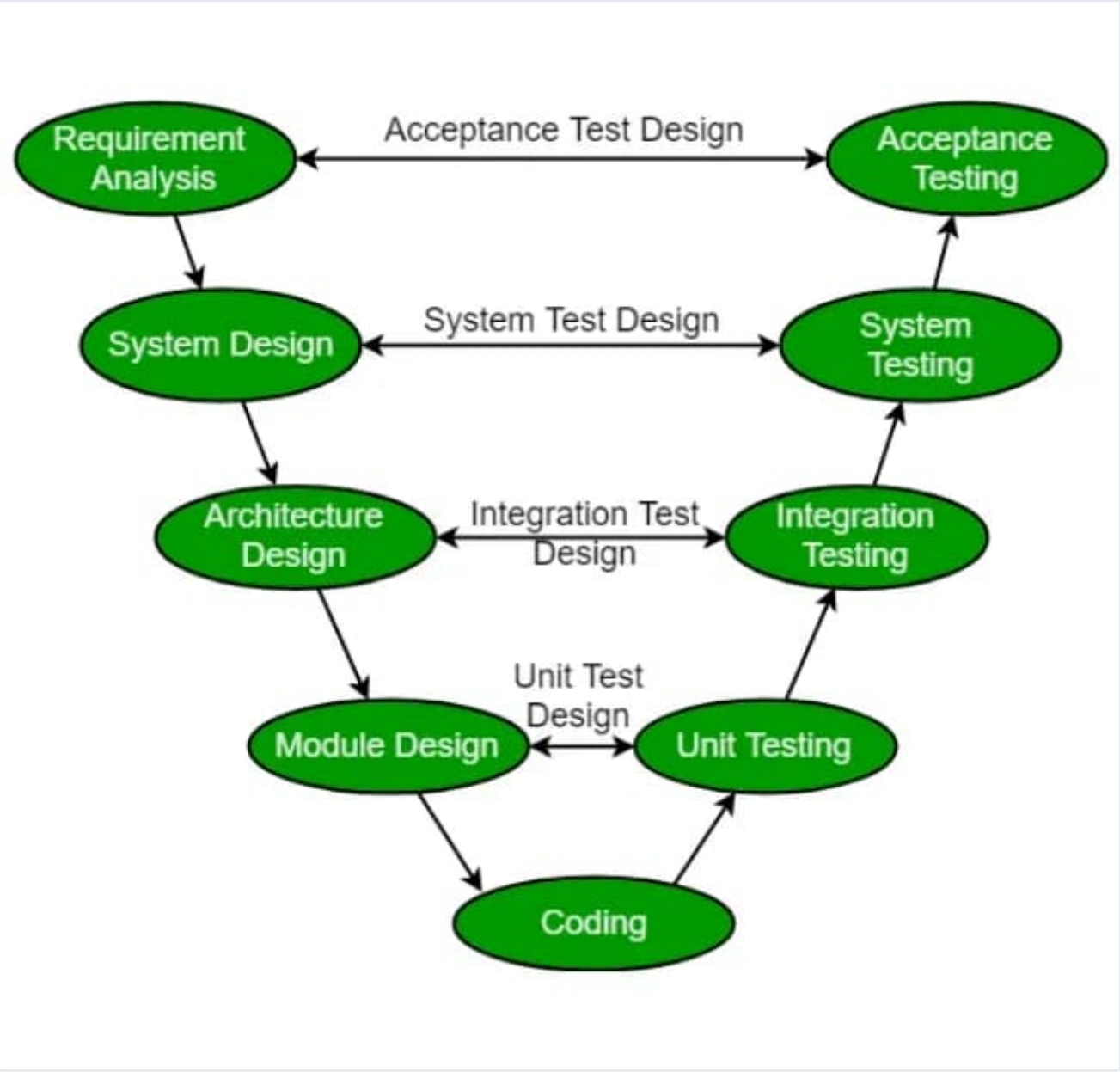
The classic model where phases flow sequentially. You cannot go back to a previous phase.



- **When to use**:
  - Requirements are very clear and fixed.
  - Technology is well-understood.
  - The project is short.
- **Advantages**: Simple, easy to manage, phases are completed one at a time.
- **Disadvantages**: High risk, no working software until the end, poor model for long and ongoing projects.

3.2 V-Model (Verification and Validation)

An extension of Waterfall where execution of processes happens in a sequential manner in a V-shape. For every single phase in the development cycle, there is a directly associated testing phase.



- **Left Side (Verification)**: Requirements -> System Design -> Arch Design -> Module Design.
- **Right Side (Validation)**: Unit Test -> Integration Test -> System Test -> Acceptance Test.
- **Advantage**: Higher quality due to early test planning.

3.3 Spiral Model (Risk-Driven)

Combines iterative nature of prototyping with controlled aspects of Waterfall. It is organized in spirals, each loop representing a phase of the software process.

1.Define project goals and scope
2.Identify and mitigate project risks
3.Develop and code software incrementally
4.Assess progress and adjust plans

- **4 Quadrants**:
  1. **Objective Setting**: Determine goals.
  2. **Risk Assessment**: Analyze risks (technical, cost).
  3. **Development & Validation**: Build and test.
  4. **Planning**: Plan the next spiral.
- **Best For**: Large, expensive, and complicated projects (e.g., NASA software).

3.4 Agile Model (Iterative & Incremental)

Agile breaks the product into small incremental builds. These builds are provided in iterations (Sprints).

- **Manifesto Principles**:
  - Individuals and interactions over processes and tools.
  - Working software over comprehensive documentation.
  - Customer collaboration over contract negotiation.
  - Responding to change over following a plan.
- **Scrum Framework**:
  - **Sprint**: 2-4 week cycle.
  - **Artifacts**: Product Backlog, Sprint Backlog, Burndown Charts.
  - **Ceremonies**: Daily Standup, Sprint Review, Retrospective.

Feature	Waterfall	Spiral	Agile
Flexibility	Rigid	Moderate	High
Risk Handling	Low	High	Moderate
Customer Interaction	Low	Moderate	High
Cost	Low	High	Moderate
Documentation	Heavy	Heavy	Light

4. Practice Questions

Multiple Choice Questions

Q1. Which phase of SDLC involves gathering user needs?

- A) Design
- B) Testing
- C) Requirement Analysis
- D) Maintenance

Answer: C

Q2. Which model is best suited for projects with high risk?

- A) Waterfall
- B) Spiral
- C) V-Model
- D) Iterative

Answer: B

Q3. In which model is "Sprint" a key concept?

- A) Waterfall
- B) Agile
- C) Spiral
- D) V-Model

Answer: B

Q4. What does SRS stand for?

- A) System Requirement System
- B) Software Requirement Specification
- C) Software Resource System

- D) System Resource Specification

Answer: B

Q5. Which model emphasizes Verification and Validation phases running in parallel?

- A) Waterfall
- B) Agile
- C) V-Model
- D) Spiral

Answer: C

Q6. Which SDLC phase results in the "Design Document Specification" (DDS)?

- A) Requirement Analysis
- B) Design
- C) Implementation
- D) Testing

Answer: B

Q7. What is a major disadvantage of the Waterfall model?

- A) It is difficult to understand
- B) It requires too much documentation
- C) It is inflexible to changes once a phase is complete
- D) It is not suitable for small projects

Answer: C

Q8. In the Spiral Model, what is the primary purpose of the "Risk Analysis" phase?

- A) To write code
- B) To identify and resolve potential risks
- C) To release the software to users
- D) To gather initial requirements

Answer: B

Q9. Which of the following is NOT a phase in the classic SDLC?

- A) Requirement Analysis
- B) Coding
- C) Marketing
- D) Maintenance

Answer: C

Q10. The "Increment" in Scrum refers to:

- A) The increase in team size
- B) The sum of all Product Backlog items completed during a Sprint
- C) The salary raise for developers
- D) The number of bugs fixed

Answer: B

Short Answer Questions

1. **Define Software Engineering.**
   Answer: Software Engineering is the systematic application of engineering principles to the development, operation, and maintenance of software to ensure quality, reliability, and efficiency.
2. **Explain the Waterfall Model.**
   Answer: The Waterfall Model is a linear SDLC model where development flows sequentially through phases like Requirements, Design, Implementation, Testing, and Maintenance. Each phase must complete before the next begins.
3. **What is the main advantage of the Agile Model?**
   Answer: The main advantage of Agile is its flexibility and adaptability to change. It allows for frequent reassessment of plans and requirements, ensuring the final product meets customer needs even if they evolve.
4. **Describe the Spiral Model's focus.**
   Answer: The Spiral Model focuses heavily on Risk Analysis. It combines iterative development with the controlled aspects of the Waterfall model, repeatedly moving through phases of planning, risk analysis, engineering, and evaluation.
5. **List the 6 main phases of SDLC.**
   Answer: Requirement Analysis, Design, Implementation (Coding), Testing, Deployment, Maintenance.
6. **Differentiate between Verification and Validation in the V-Model.**
   Answer: Verification asks "Are we building the product right?" (checking against requirements/design). Validation asks "Are we building the right product?" (checking if it meets user needs).
7. **What is a "Sprint" in Agile/Scrum?**
   Answer: A Sprint is a short, time-boxed period (usually 2-4 weeks) during which a Scrum team works to complete a set amount of work and create a shippable product increment.

FOUNDATIONS OF DESIGN & ARCHITECTURE

1. Software Design Concepts

Software design is the process of defining software methods, functions, objects, and the overall structure and interaction of your code so that the resulting functionality will satisfy your users' requirements.

1.1 Abstraction

Abstraction is the process of hiding complex implementation details and showing only the essential features of the object. It reduces complexity and allows developers to focus on interactions at a higher level.

- **Procedural Abstraction**: Hiding details of a function/method. For example, calling `sort(list)` without knowing if it uses QuickSort or MergeSort.
- **Data Abstraction**: Hiding details of data structures. For example, using a `Stack` class without knowing if it's implemented with an Array or Linked List.

1.2 Modularity

Modularity is the degree to which a system's components may be separated and recombined. It divides the software into distinct modules that are developed and tested independently.

- **Benefits**:
  - **Reusability**: Modules can be used in other projects.
  - **Maintainability**: Easier to fix bugs in a small module than a large monolith.
  - **Parallel Development**: Teams can work on different modules simultaneously.

1.3 Cohesion and Coupling (The Golden Rule)

The ultimate goal of software design is **High Cohesion** and **Low Coupling**.

Cohesion (Internal Strength)

Cohesion measures how strongly related the responsibilities of a single module are.

- **Functional Cohesion (Best)**: Every part of the module contributes to a single well-defined task (e.g., `Math.sin()`).
- **Sequential Cohesion**: Output of one part is input to another.
- **Logical Cohesion (Worst)**: Parts are related only logically (e.g., a utility class with `printDocument`, `calculateTax`, and `sendEmail`).

Coupling (External Dependency)

Coupling measures how dependent one module is on another.

- **Data Coupling (Best)**: Modules share data through parameters (e.g., passing an integer).
- **Control Coupling**: One module controls the flow of another (e.g., passing a "flag" boolean).
- **Content Coupling (Worst)**: One module modifies the internal data of another directly.

---

2. Design Principles (SOLID)

SOLID is an acronym for five design principles intended to make software designs more understandable, flexible, and maintainable.

2.1 S - Single Responsibility Principle (SRP)

"A class should have only one reason to change."

- **Explanation**: A class should have only one job. If a class handles both database operations and HTML rendering, it violates SRP.
- **Violation**:

```
class User {
    void saveToDb() { ... }
    void generateReport() { ... } // Violation: Mixing logic with repor
}
```

- **Fix**: Split into `UserRepository` (db) and `UserReportGenerator` (reporting).

2.2 O - Open/Closed Principle (OCP)

"Software entities should be open for extension, but closed for modification."

- **Explanation**: You should be able to add new functionality without changing existing code. This is usually achieved via **Polymorphism**.
- **Example**: Instead of a `switch` statement checking for `Dog`, `Cat`, `Bird` to call `makeSound()`, use an interface `Animal` with `makeSound()`. To add a `Lion`, you just create a new class; you don't touch the existing logic.

2.3 L - Liskov Substitution Principle (LSP)

"Subtypes must be substitutable for their base types."

- **Explanation**: If `B` is a subclass of `A`, you should be able to replace `A` with `B` without breaking the program.
- **Classic Violation**: The Square-Rectangle problem. If `Rectangle` has `setWidth` and `setHeight`, and `Square` overrides them to force width=height, then code expecting a `Rectangle` might fail when

given a `Square`.

2.4 I - Interface Segregation Principle (ISP)

"Clients should not be forced to depend on interfaces they do not use."

- **Explanation**: Many client-specific interfaces are better than one general-purpose interface.
- **Example**:
  - *Bad*: `interface Worker { work(); eat(); }` (Robots implement this but throw error on `eat()`).
  - *Good*: `interface Workable { work(); }`, `interface Eatable { eat(); }`.

2.5 D - Dependency Inversion Principle (DIP)

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

- **Explanation**: Don't hardcode dependencies. Use interfaces.
- **Example**:
  - *Bad*: `Store` depends directly on `StripePaymentService`.
  - *Good*: `Store` depends on `PaymentProcessor` interface. `StripePaymentService` implements it. This allows switching to PayPal easily.

---

3. GRASP Principles

GRASP (General Responsibility Assignment Software Patterns) helps in assigning responsibilities to classes.

1. **Creator**: Assign responsibility to create `B` to class `A` if:
   - A contains B.
   - A records B.
   - A closely uses B.
2. **Information Expert**: Assign responsibility to the class that has the information necessary to fulfill it. (e.g., `Sale` class should calculate total because it knows the `LineItems`).
3. **Controller**: The first object beyond the UI layer that receives and coordinates a system operation. It delegates work to other objects.
4. **Low Coupling**: Assign responsibilities so coupling remains low.
5. **High Cohesion**: Assign responsibilities so cohesion remains high.

---

4. Software Architecture

Software Architecture refers to the high-level structure of a software system. It is the blueprint of the system, defining the components and their relationships.

Architecture vs. Design

- **Architecture**: High-level, strategic decisions (e.g., "We will use Microservices", "We will use Oracle DB"). Hard to change.
- **Design**: Low-level, tactical decisions (e.g., "We will use the Factory Pattern here"). Easier to change.

Architectural Design Issues

1. **Performance**: Response time, throughput.
2. **Scalability**: Ability to handle growth (vertical vs. horizontal scaling).
3. **Reliability**: Fault tolerance, availability (uptime).
4. **Security**: Authentication, authorization, encryption.
5. **Maintainability**: Ease of modification and bug fixing.

---

5. Practice Questions

Multiple Choice Questions

Q1. Which principle states "A class should have only one reason to change"?

- A) Open/Closed Principle
- B) Single Responsibility Principle
- C) Dependency Inversion Principle
- D) Liskov Substitution Principle

Answer: B

Q2. High Cohesion means:

- A) Modules are heavily dependent on each other
- B) Elements within a module are closely related
- C) The system is difficult to maintain
- D) Modules are large and complex

Answer: B

Q3. Which GRASP pattern suggests assigning responsibility to the class that has the necessary information?

- A) Creator
- B) Controller

- C) Information Expert
- D) Low Coupling

Answer: C

Q4. In SOLID, what does 'O' stand for?

- A) Object-Oriented
- B) Open/Closed
- C) Output/Input
- D) Overload/Override

Answer: B

Q5. Software Architecture focuses on:

- A) Writing algorithms
- B) High-level structure of the system
- C) Designing individual functions
- D) Testing the code

Answer: B

Q6. Which principle suggests that "Clients should not be forced to depend on interfaces they do not use"?

- A) Single Responsibility Principle
- B) Interface Segregation Principle
- C) Liskov Substitution Principle
- D) Dependency Inversion Principle

Answer: B

Q7. What is the main goal of "Modularity"?

- A) To make the code run faster
- B) To divide software into distinct, independent components
- C) To combine all functions into one file
- D) To increase coupling between modules

Answer: B

Q8. Which SOLID principle is violated if a `Bird` class has a `fly()` method, but `Penguin` inherits from it and throws an error?

- A) SRP
- B) OCP

- C) LSP
- D) DIP

Answer: C

Q9. In GRASP, which pattern assigns responsibility to the object that creates instances of another class?

- A) Creator
- B) Controller
- C) Expert
- D) Polymorphism

Answer: A

Q10. "Procedural Abstraction" involves:

- A) Hiding data structures
- B) Hiding implementation details of a function
- C) Hiding the entire class
- D) Hiding the database

Answer: B

Short Answer Questions

1. **Explain the difference between Cohesion and Coupling.**
   Answer: Cohesion refers to how closely related the elements are *within* a single module (we want this High). Coupling refers to how dependent different modules are *on each other* (we want this Low).
2. **Describe the Open/Closed Principle.**
   Answer: Software entities (classes, modules, functions) should be open for extension (adding new functionality) but closed for modification (changing existing code). This prevents breaking existing code when adding features.
3. **What is the Dependency Inversion Principle?**
   Answer: It states that high-level modules should not depend on low-level modules; both should depend on abstractions (interfaces). This decouples the code and makes it easier to change implementations.
4. **Define Software Architecture.**
   Answer: Software Architecture is the high-level structure of a software system, defining its components, their relationships, and the principles guiding its design and evolution.
5. **List three Architectural Design Issues.**
   Answer: Scalability, Performance, Security (also Reliability, Maintainability).
6. **Explain the Liskov Substitution Principle (LSP) with an example.**
   Answer: LSP states that objects of a superclass should be replaceable with objects of its subclasses

without breaking the application. For example, if `Square` is a subclass of `Rectangle`, it should behave like a `Rectangle` in all contexts; if setting the width of a `Square` unexpectedly changes its height (unlike a `Rectangle`), it violates LSP.

7. **What is the "Controller" pattern in GRASP?**
   Answer: The Controller pattern assigns the responsibility of handling system events to a non-UI class that represents the overall system or a use case scenario. It acts as the first point of contact for handling user actions from the UI layer.

MODELING & OBJECT-ORIENTED DESIGN (OOD)

1. Object-Oriented Design (OOD) with UML

Object-Oriented Design (OOD) involves planning a system of interacting objects for the purpose of solving a software problem. The **Unified Modeling Language (UML)** is the standard visual modeling language used to describe, specify, design, and document these systems.

1.1 Why Model?

Modeling is a way of thinking about problems using models organized around real-world ideas.

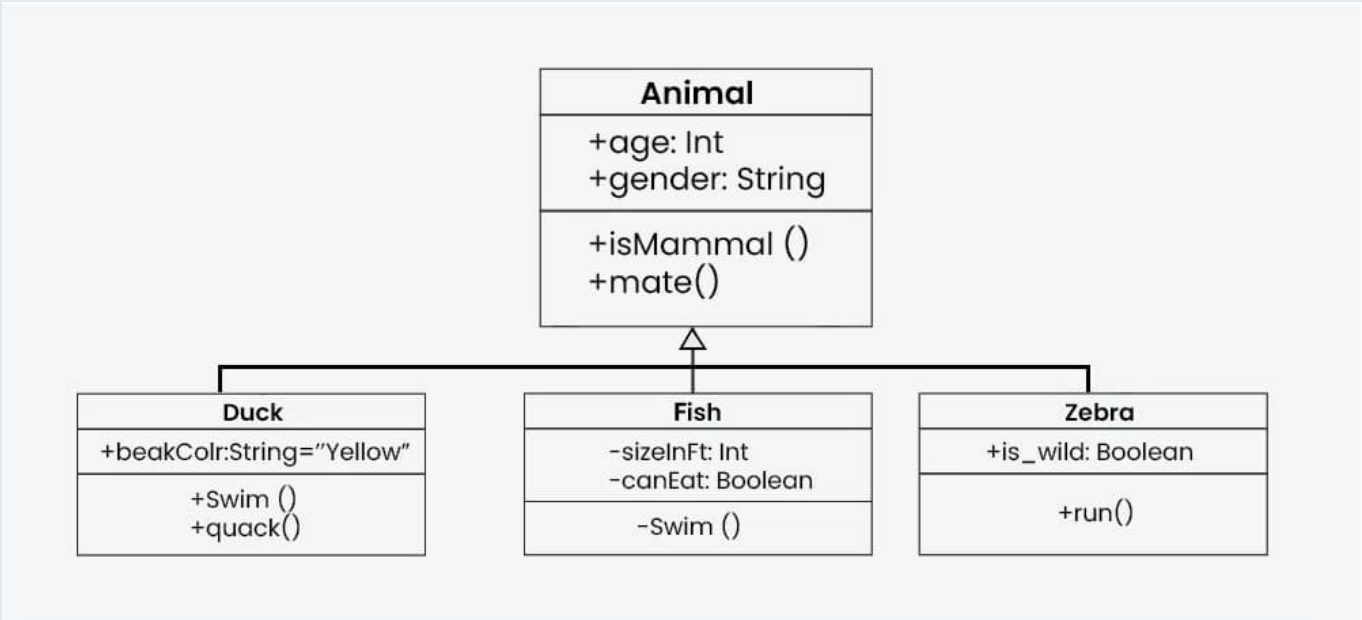
- **Visualization**: See the big picture.
- **Specification**: Precise details before coding.
- **Documentation**: For future maintenance.
- **Communication**: Common language for developers.

1.2 Structural Diagrams (Static View)

Structural diagrams depict the static structure of the system (the "bones").

Class Diagram

The most common diagram. It shows classes, their attributes, methods, and relationships.



Class Diagram example

- **Class**: A blueprint for objects. Represented by a rectangle with 3 compartments (Name, Attributes, Methods).
- **Relationships**:
  - **Association**: A basic link (e.g., Teacher teaches Student).
  - **Inheritance (Generalization)**: "Is-a" relationship (e.g., Cat is an Animal).
  - **Aggregation**: "Has-a" relationship (weak). The part can exist without the whole (e.g., Class has Students).
  - **Composition**: "Part-of" relationship (strong). The part *cannot* exist without the whole (e.g., House has Rooms).

```
classDiagram
    class Animal {
        +String name
        +int age
        +eat()
        +sleep()
    }
    class Dog {
        +bark()
    }
    class Cat {
        +meow()
    }
    Animal <|-- Dog : Inheritance
    Animal <|-- Cat : Inheritance

    class Car {
        +Engine engine
    }
    class Engine {
        +start()
    }
    Car *-- Engine : Composition (Strong)
```

1.3 Behavioral Diagrams (Dynamic View)

Behavioral diagrams depict the dynamic behavior of the system (the "muscle").

Sequence Diagram

Shows how objects interact with each other in a specific time sequence.

- **Lifeline**: Vertical dashed line representing the object's life.
- **Message**: Horizontal arrow representing communication.

- **Activation Bar**: Rectangle on lifeline showing when object is active.

```
sequenceDiagram
    participant User
    participant ATM
    participant Bank

    User->>ATM: Insert Card
    ATM->>Bank: Verify Card
    Bank-->>ATM: Card OK
    ATM-->>User: Request PIN
    User->>ATM: Enter PIN
    ATM->>Bank: Verify PIN
    Bank-->>ATM: PIN OK
    ATM-->>User: Show Menu
```

Use Case Diagram

Shows the interactions between a system and users (actors).

- **Actor**: Stick figure representing a user or external system.
- **Use Case**: Oval representing a function (e.g., "Login").
- **System Boundary**: Box around use cases.

---

2. Object Design

Object design is the process of adding details to the class definitions. It transforms the analysis model into a design model.

2.1 The Process

1. **Identify Classes**: Nouns in requirements often become classes.
2. **Identify Attributes**: Characteristics of the classes.
3. **Identify Methods**: Verbs in requirements often become methods.
4. **Define Relationships**: How classes interact.

2.2 Visibility Modifiers

Visibility determines who can access the members of a class.

Symbol	Visibility	Description
+	**Public**	Accessible from anywhere.
-	**Private**	Accessible only within the class itself. (Encapsulation)
#	**Protected**	Accessible within the class and its subclasses.
~	**Package**	Accessible within the same package (default in Java).

3. State Machine Diagrams

A **State Machine Diagram** (or State Chart) describes the behavior of a single object or system in response to events. It is crucial for systems with complex lifecycles (e.g., an Order, a Traffic Light).

Key Elements:

- **State**: A condition or situation during the life of an object (e.g., "Pending", "Shipped").
- **Transition**: Movement from one state to another triggered by an event.
- **Event**: Something that happens (e.g., "Payment Received").
- **Guard Condition**: A boolean condition that must be true for transition (e.g., `[balance > 0]`).

Example: Order Processing

```
stateDiagram-v2
    [*] --> New
    New --> Processing : Payment Verified
    Processing --> Shipped : Order Packed
    Shipped --> Delivered : Customer Received
    New --> Cancelled : User Cancelled
    Processing --> Cancelled : Out of Stock
    Delivered --> [*]
    Cancelled --> [*]
```

4. Exploring Inheritance

Inheritance is a mechanism where a new class (subclass/child) derives properties and behavior from an existing class (superclass/parent).

4.1 Types of Inheritance

1. **Single Inheritance**: A class inherits from one superclass. (Supported by Java/C#).
2. **Multilevel Inheritance**: A chain (A -> B -> C).

3. **Hierarchical Inheritance**: Multiple classes inherit from one superclass (Animal -> Cat, Dog).
4. **Multiple Inheritance**: A class inherits from multiple superclasses (Supported in C++, Python; *not* in Java to avoid "Diamond Problem").

4.2 Composition over Inheritance

"Favor Composition over Inheritance."

- **Inheritance (Is-a)**: Rigid. Changes in parent affect all children. Breaks encapsulation.
- **Composition (Has-a)**: Flexible. You can change behavior at runtime by swapping components.

---

5. Model-Driven Development (MDD)

Model-Driven Development (MDD) is a paradigm where models are the primary artifacts, not code.

- **Concept**: Developers create high-level models (like UML), and tools automatically generate the code.
- **MDA (Model Driven Architecture)**: An OMG standard.
  - **CIM (Computation Independent Model)**: Domain / Business view.
  - **PIM (Platform Independent Model)**: Logical view (UML).
  - **PSM (Platform Specific Model)**: Technical view (Java/C# specific).
  - **Code**: Generated from PSM.

Advantages:

- **Productivity**: Write less boilerplate code.
- **Quality**: Generated code is consistent and error-free.
- **Portability**: Regenerate code for a different platform from the same model.

---

6. Practice Questions

Multiple Choice Questions

Q1. Which diagram shows the static structure of a system?

- A) Sequence Diagram
- B) Class Diagram
- C) State Machine Diagram
- D) Use Case Diagram

Answer: B

Q2. In a Sequence Diagram, what does the vertical dimension represent?

- A) Objects
- B) Messages
- C) Time
- D) Architecture

Answer: C

Q3. Which relationship represents "Part-of" where the part cannot exist without the whole?

- A) Association
- B) Aggregation
- C) Composition
- D) Inheritance

Answer: C

Q4. What is the main goal of Model-Driven Development?

- A) To write more code manually
- B) To use models as the primary source for generating code
- C) To avoid using UML
- D) To focus only on testing

Answer: B

Q5. Which symbol represents Private visibility in UML?

- A) `+`
- B) `-`
- C) `#`
- D) `~`

Answer: B

Q6. Which type of inheritance allows a class to inherit from multiple superclasses?

- A) Single Inheritance
- B) Multilevel Inheritance
- C) Hierarchical Inheritance
- D) Multiple Inheritance

Answer: D

Q7. In a Class Diagram, what does the diamond symbol represent?

- A) Inheritance

- B) Aggregation or Composition
- C) Dependency
- D) Realization

Answer: B

Q8. Which diagram is best for modeling the lifecycle of an object?

- A) Class Diagram
- B) Sequence Diagram
- C) State Machine Diagram
- D) Component Diagram

Answer: C

Q9. What does PIM stand for in Model-Driven Architecture?

- A) Platform Independent Model
- B) Platform Integrated Model
- C) Program Independent Model
- D) Platform Internal Model

Answer: A

Q10. Which relationship is described as "Is-a"?

- A) Association
- B) Aggregation
- C) Composition
- D) Inheritance

Answer: D

Short Answer Questions

1. **Differentiate between Aggregation and Composition.**
   Answer: Aggregation is a "weak" relationship where the child can exist independently of the parent (e.g., Class and Student). Composition is a "strong" relationship where the child cannot exist without the parent (e.g., House and Room).
2. **What is the purpose of a State Machine Diagram?**
   Answer: It models the dynamic behavior of an object by showing the different states it can be in and how it transitions between them based on events.
3. **Explain the concept of "Composition over Inheritance".**
   Answer: It is a design principle suggesting that classes should achieve polymorphic behavior and

code reuse by containing instances of other classes (composition) rather than inheriting from them. This reduces coupling and increases flexibility.

4. **List three types of UML diagrams.**
   Answer: Class Diagram, Sequence Diagram, Use Case Diagram (also State Machine, Activity, Component).

5. **What is a Use Case Diagram used for?**
   Answer: It is used to capture the functional requirements of a system by showing the interactions between users (actors) and the system's use cases (functionalities).

6. **What is Model-Driven Architecture (MDA)?**
   Answer: MDA is an approach to software design that separates the specification of the system's functionality from its implementation on a specific technology platform. It uses Platform Independent Models (PIMs) which are then transformed into Platform Specific Models (PSMs).

7. **Describe the role of an "Actor" in a Use Case Diagram.**
   Answer: An Actor represents a role played by a user or any other system that interacts with the subject system. Actors are external to the system and initiate or participate in use cases.

ARCHITECTURAL STYLES, PATTERNS & QUALITIES

1. Architectural Structures & Styles

An **Architectural Style** defines a family of systems in terms of a pattern of structural organization. It provides a vocabulary of components and connectors, and a set of constraints on how they can be combined.

1.1 Layered Architecture (N-Tier)

Organizes the system into horizontal layers, where each layer provides services to the layer above it and uses services from the layer below.

- **Structure**:
  1. **Presentation Layer**: UI and user interaction.
  2. **Business Logic Layer**: Core functionality and rules.
  3. **Data Access Layer**: Communication with the database.
  4. **Database**: Physical storage.
- **Strict vs. Relaxed**: In strict layering, a layer can only call the one directly below. In relaxed, it can call any layer below.
- **Pros**: Separation of concerns, testability, maintainability.
- **Cons**: Performance overhead (data must pass through all layers), "Sinkhole" anti-pattern (layers that just pass data without adding value).

1.2 Client-Server Architecture

Distributes tasks between the providers of a resource or service (servers) and service requesters (clients).

- **Thin Client**: Most logic is on the server (e.g., Web App).
- **Thick Client**: Most logic is on the client (e.g., Desktop Game).
- **Pros**: Centralized control, easier maintenance of data.
- **Cons**: Server is a single point of failure, network congestion.

1.3 Peer-to-Peer (P2P)

A decentralized model where each node (peer) acts as both a client and a server.

- **Examples**: BitTorrent, Blockchain.

- **Pros**: Robustness (no single point of failure), scalability.
- **Cons**: Complex management, security challenges, no centralized data authority.

1.4 Microservices Architecture

Structuring an application as a collection of loosely coupled services.

- **Characteristics**:
  - Modeled around business domains (e.g., Order Service, User Service).
  - Independently deployable.
  - Decentralized data management (each service has its own DB).
- **Pros**: Scalability, flexibility in technology stack, fault isolation.
- **Cons**: Complexity in deployment, distributed system challenges (network latency, consistency).

---

2. Architectural Patterns vs. Design Patterns

It is crucial to distinguish between these two concepts.

Feature	Architectural Pattern	Design Pattern
Scope	System-wide	Component/Class level
Example	MVC, Microservices, Layered	Singleton, Factory, Observer
Impact	Affects the entire application structure	Affects a specific part of the code
Abstraction	High-level	Low-level

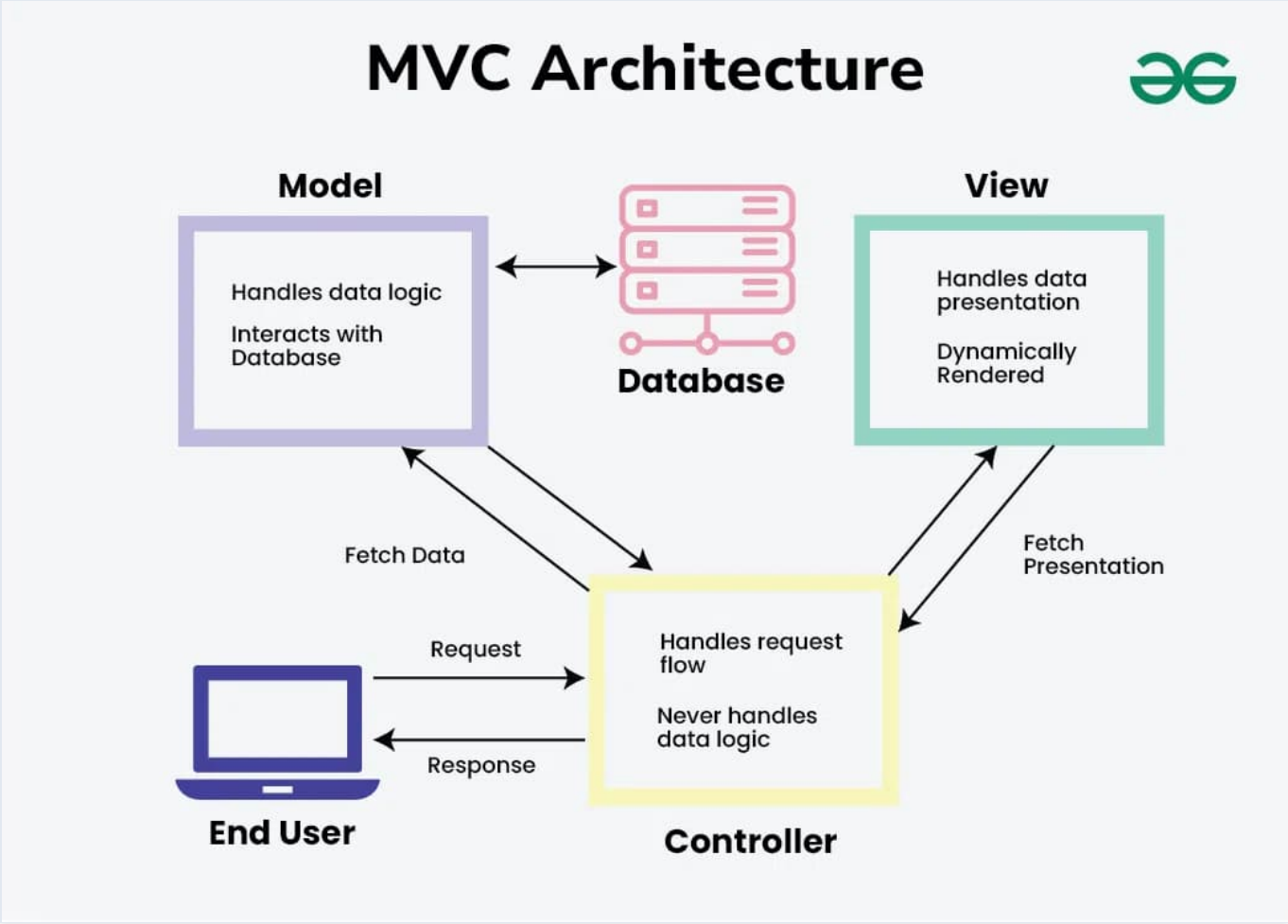
2.1 Common Design Patterns (Gang of Four)

1. **Creational Patterns**: Deal with object creation mechanisms.
   - **Singleton**: Ensures a class has only one instance (e.g., Database Connection).
   - **Factory**: Creates objects without specifying the exact class.
2. **Structural Patterns**: Deal with object composition.
   - **Adapter**: Allows incompatible interfaces to work together (e.g., XML to JSON adapter).
   - **Facade**: Provides a simplified interface to a complex system.
3. **Behavioral Patterns**: Deal with communication between objects.
   - **Observer**: A subscription mechanism to notify multiple objects about events (e.g., Event Listeners).
   - **Strategy**: Defines a family of algorithms and makes them interchangeable.

---

3. Interactive Systems with MVC Architecture

Model-View-Controller (MVC) is an architectural pattern commonly used for developing user interfaces. It divides an application into three interconnected parts.



3.1 Components

1. **Model**: Manages the data, logic, and rules of the application. It is independent of the user interface. It notifies the View of changes.
2. **View**: The user interface (what the user sees). It displays data from the Model.
3. **Controller**: Accepts input and converts it to commands for the Model or View.

3.2 Interaction Flow

1. User interacts with the **View** (clicks a button).
2. **Controller** handles the input event.
3. **Controller** notifies the **Model** to update state (e.g., "Delete User").
4. **Model** updates and notifies the **View** (or View polls Model).
5. **View** refreshes to show the new state.

3.3 Variants

- **MVP (Model-View-Presenter)**: Used in Windows Forms/Android. Presenter is middleman; View is passive.

- **MVVM (Model-View-ViewModel)**: Used in Angular/WPF. Two-way data binding between View and ViewModel.

---

4. Architectural & Design Qualities

Quality attributes (also known as non-functional requirements) measure "how well" the system performs its functions.

4.1 Key Quality Attributes

1. **Scalability**: The ability to handle growing amounts of work (e.g., adding more users).
2. **Maintainability**: The ease with which a product can be maintained (fix bugs, add features).
3. **Performance**: Responsiveness (latency) and resource usage (throughput).
4. **Security**: Protection against unauthorized access and data breaches.
5. **Reliability**: Ability to perform functions under stated conditions for a specified time (Availability/Uptime).
6. **Usability**: Ease of use and learnability.

4.2 Quality Tactics

Tactics are specific design decisions used to achieve quality attributes.

- **For Availability**:
  - **Redundancy**: Backup servers (Active/Passive).
  - **Heartbeat**: Pinging servers to check health.
  - **Exception Handling**: Graceful failure instead of crashing.
- **For Performance**:
  - **Caching**: Storing frequently used data in memory.
  - **Concurrency**: Processing requests in parallel.
  - **Load Balancing**: Distributing traffic across servers.
- **For Security**:
  - **Authentication**: Verifying identity.
  - **Authorization**: Verifying permissions.
  - **Encryption**: Protecting data at rest and in transit.

---

5. Practice Questions

Multiple Choice Questions

Q1. Which architectural style organizes the system into horizontal levels?

- A) Client-Server
- B) Layered
- C) Microservices
- D) Peer-to-Peer

Answer: B

Q2. In MVC, which component handles the business logic and data?

- A) Model
- B) View
- C) Controller
- D) User

Answer: A

Q3. Which pattern ensures a class has only one instance?

- A) Factory
- B) Adapter
- C) Singleton
- D) Observer

Answer: C

Q4. "Caching" is a tactic primarily used to improve which quality attribute?

- A) Security
- B) Maintainability
- C) Performance
- D) Usability

Answer: C

Q5. Which is a behavioral design pattern?

- A) Singleton
- B) Facade
- C) Observer
- D) Adapter

Answer: C

Q6. Which quality attribute refers to the ability of the system to handle increased load?

- A) Maintainability

- B) Scalability
- C) Usability
- D) Security

Answer: B

Q7. In a Peer-to-Peer (P2P) architecture:

- A) There is a central server controlling everything
- B) Each node acts as both client and server
- C) All data is stored in a single database
- D) It is impossible to share files

Answer: B

Q8. Which pattern provides a simplified interface to a complex subsystem?

- A) Adapter
- B) Facade
- C) Strategy
- D) Observer

Answer: B

Q9. "Redundancy" is a tactic primarily used to improve:

- A) Performance
- B) Availability/Reliability
- C) Usability
- D) Modifiability

Answer: B

Q10. Which architectural style is best suited for highly decoupled, independently deployable services?

- A) Monolithic
- B) Layered
- C) Microservices
- D) Client-Server

Answer: C

Short Answer Questions

1. **Explain the Microservices Architecture.**
   Answer: It is an architectural style where an application is structured as a collection of small,

autonomous services, modeled around a business domain. Each service runs in its own process and communicates with lightweight mechanisms (often HTTP).

2. **What is the difference between Architectural Patterns and Design Patterns?**
   Answer: Architectural patterns (like MVC) address system-wide structure and organization, while design patterns (like Singleton) address specific, localized coding problems within components.
3. **Describe the role of the Controller in MVC.**
   Answer: The Controller acts as an intermediary between the View and the Model. It receives user input from the View, processes it, and updates the Model accordingly.
4. **List three tactics to improve Availability.**
   Answer: Redundancy (backup servers), Heartbeat (monitoring health), and Exception Handling (graceful failure).
5. **What is the Layered Architecture style?**
   Answer: A style that organizes the system into layers (e.g., Presentation, Business, Data), where each layer provides services to the layer above it and uses services from the layer below.
6. **Define the "Observer" design pattern.**
   Answer: The Observer pattern defines a one-to-many dependency between objects so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically. It is commonly used in event handling systems.
7. **What are the pros and cons of Client-Server Architecture?**
   Answer: **Pros**: Centralized control, easier maintenance, scalable. **Cons**: The server is a single point of failure, and network congestion can occur if too many clients request services simultaneously.

SPECIALIZED DESIGN LAYERS

1. User Interface (UI) Design

User Interface (UI) Design is the process of building interfaces in software or computerized devices, focusing on looks or style. It aims to create interfaces which users find easy to use and pleasurable.

1.1 UI vs. UX Design

While often used interchangeably, UI and UX are distinct disciplines:

- **UX (User Experience) Design**: Focuses on the overall feel of the experience. It involves research, testing, and understanding user needs. It answers the question: "Is this product useful and easy to use?"
- **UI (User Interface) Design**: Focuses on the look and layout. It involves colors, typography, buttons, and visual hierarchy. It answers the question: "Does this product look good and function intuitively?"

Feature	UX Design	UI Design
Focus	Functionality & Experience	Visuals & Aesthetics
Goal	Solve user problems	Make the interface beautiful
Tools	Wireframes, Personas, User Flows	Mockups, Prototypes, Style Guides
Output	Wireframes, Sitemaps	High-fidelity screens, Icons

1.2 Golden Rules of Interface Design (Shneiderman)

Ben Shneiderman proposed 8 golden rules that are widely accepted as the standard for good UI design:

1. **Strive for Consistency**: Consistent sequences of actions should be required in similar situations. Terminology (e.g., "Save" vs. "Write") should be identical in prompts, menus, and help screens.
2. **Enable Frequent Users to Use Shortcuts**: As the frequency of use increases, so do the user's desires to reduce the number of interactions. Abbreviations, function keys, hidden commands, and macro facilities are very helpful to an expert user.
3. **Offer Informative Feedback**: For every operator action, there should be some system feedback. For frequent and minor actions, the response can be modest, while for infrequent and major actions, the response should be more substantial.
4. **Design Dialog to Yield Closure**: Sequences of actions should be organized into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the operators the satisfaction of accomplishment.

5. **Offer Simple Error Handling**: As much as possible, design the system so the user cannot make a serious error. If an error is made, the system should detect the error and offer simple, comprehensible mechanisms for handling the error.
6. **Permit Easy Reversal of Actions**: This feature relieves anxiety, since the user knows that errors can be undone. It encourages exploration of unfamiliar options.
7. **Support Internal Locus of Control**: Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.
8. **Reduce Short-Term Memory Load**: The limitation of human information processing in short-term memory requires that displays be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions.

---

2. Persistent Layer Design

The **Persistence Layer** is the part of the software architecture that deals with storing, retrieving, and updating data in a permanent storage system (usually a database). It separates the business logic from the details of database access.

2.1 The Need for Persistence

In object-oriented programming, objects exist in memory (RAM). When the program ends or the computer shuts down, this memory is cleared. To save data permanently, we need to "persist" these objects to a disk-based database.

2.2 Object-Relational Mapping (ORM)

One of the biggest challenges in persistence design is the **Impedance Mismatch**.

- **Object-Oriented World**: Uses classes, objects, inheritance, and references.
- **Relational Database World**: Uses tables, rows, columns, and foreign keys.

ORM (Object-Relational Mapping) is a technique (and often a library) that automatically bridges this gap. It maps objects in code to tables in the database.

How ORM Works:

1. **Class to Table**: A class `Student` maps to a table `students`.
2. **Property to Column**: A property `name` maps to a column `name`.
3. **Object to Row**: An instance of `Student` maps to a row in the `students` table.

Examples of ORM Frameworks:

- **Hibernate** (Java)
- **Entity Framework** (.NET)

- **Sequelize** (Node.js)
- **Django ORM** (Python)

2.3 Data Access Object (DAO) Pattern

The **DAO Pattern** is a structural pattern that isolates the application/business layer from the persistence layer using an abstract API.

- **Interface**: Defines standard operations (Create, Read, Update, Delete - CRUD).
- **Implementation**: Contains the specific code to access the database (SQL queries).
- **Benefit**: You can change the underlying database (e.g., from MySQL to MongoDB) without changing the business logic, just by creating a new DAO implementation.

```
classDiagram
    class BusinessService {
        +registerUser()
    }
    class UserDAO {
        <<interface>>
        +save(User)
        +findById(int)
    }
    class UserDAOImpl {
        +save(User)
        +findById(int)
    }
    class Database {
        +executeQuery()
    }

    BusinessService --> UserDAO
    UserDAO <|.. UserDAOImpl
    UserDAOImpl --> Database
```

---

3. Web Applications Design

Designing for the web introduces specific constraints and requirements compared to desktop applications.

3.1 Statelessness of HTTP

The web is built on HTTP, which is a **stateless protocol**. This means the server treats every request as a new, independent transaction. It does not "remember" previous requests from the same user.

Design Challenge: How to maintain a user's session (e.g., keeping them logged in)?
Solutions:

- **Cookies**: Small pieces of data stored on the client.
- **Sessions**: Server-side storage referenced by a session ID in a cookie.
- **Tokens (JWT)**: Self-contained tokens sent with every request.

3.2 Client-Side vs. Server-Side Rendering

- **Server-Side Rendering (SSR)**: The server generates the full HTML page and sends it to the browser.
  - *Pros*: Better SEO, faster initial load.
  - *Cons*: Higher server load, full page reloads on navigation.
- **Client-Side Rendering (CSR)**: The server sends a skeleton HTML and JavaScript. The browser executes JS to fetch data and build the UI.
  - *Pros*: Rich interactivity (SPA), lower server load.
  - *Cons*: Slower initial load, SEO challenges (though improving).

3.3 Responsive Web Design (RWD)

Web applications must work on devices of all sizes (desktops, tablets, phones). RWD is the approach of designing web content that adapts to the screen size.

Key Techniques:

- **Fluid Grids**: Using percentages instead of fixed pixels for widths.
- **Flexible Images**: Images that scale within their containers.
- **Media Queries**: CSS rules that apply only under certain conditions (e.g., `min-width: 768px`).

---

4. Practice Questions

Multiple Choice Questions

Q1. What is the primary focus of UX Design?

- A) Visual aesthetics and colors
- B) The overall feel and usability of the experience
- C) Writing database queries
- D) Designing server architecture

Answer: B

Q2. Which of Shneiderman's Golden Rules suggests that users should be able to undo their actions?

- A) Strive for Consistency

- B) Offer Informative Feedback
- C) Permit Easy Reversal of Actions
- D) Reduce Short-Term Memory Load

Answer: C

Q3. What problem does Object-Relational Mapping (ORM) solve?

- A) The speed of database queries
- B) The mismatch between object-oriented code and relational tables
- C) The security of user passwords
- D) The design of the user interface

Answer: B

Q4. In the DAO pattern, what does the Business Layer depend on?

- A) The Database directly
- B) The DAO Implementation
- C) The DAO Interface
- D) The SQL Queries

Answer: C (This ensures loose coupling)

Q5. HTTP is a stateless protocol. What does this mean?

- A) It cannot encrypt data
- B) It does not remember previous requests from the same client
- C) It is very slow
- D) It only works on static pages

Answer: B

Q6. Which of the following is NOT a Golden Rule of Interface Design?

- A) Strive for Consistency
- B) Make the interface look complex
- C) Offer Informative Feedback
- D) Permit Easy Reversal of Actions

Answer: B

Q7. Which rendering method is generally better for SEO?

- A) Client-Side Rendering (CSR)
- B) Server-Side Rendering (SSR)

- C) Hybrid Rendering
- D) Lazy Loading

Answer: B

Q8. What is a "Persona" in UX Design?

- A) A fictional character representing a user type
- B) The database administrator
- C) A security protocol
- D) A type of button

Answer: A

Q9. Which technique allows web content to adapt to different screen sizes?

- A) Fixed Layout
- B) Responsive Web Design
- C) Static Design
- D) Adaptive Images

Answer: B

Q10. In the context of web applications, what is a "Session"?

- A) A meeting between developers
- B) A server-side storage mechanism to maintain user state
- C) A type of database query
- D) A JavaScript function

Answer: B

Short Answer Questions

1. **Explain the difference between UI and UX design.**
   Answer: UI (User Interface) design focuses on the visual aspectsâ€”how the product looks (colors, layout, typography). UX (User Experience) design focuses on the functional aspectsâ€”how the product feels and how easy it is to use to solve a problem.
2. **What is the "Impedance Mismatch" in the context of database design?**
   Answer: It refers to the conceptual and technical difficulties in mapping object-oriented concepts (like inheritance, polymorphism, and object references) to the tabular structure of relational databases (tables, rows, foreign keys).
3. **Describe the Data Access Object (DAO) pattern and its benefits.**
   Answer: The DAO pattern abstracts the data persistence logic behind an interface. The business logic

calls methods on the interface (like `saveUser`) without knowing the details of the database implementation (SQL, file system, etc.). This makes the code easier to test and maintain.

4. **Why is "Consistency" important in UI design?**
   Answer: Consistency ensures that similar actions yield similar results and that terminology is uniform throughout the application. This reduces the user's learning curve and prevents confusion.

5. **How do web applications handle state despite HTTP being stateless?**
   Answer: Web applications use mechanisms like Cookies (client-side storage), Sessions (server-side storage linked by an ID), or Tokens (like JWTs) to maintain state and identify users across multiple requests.

6. **What are the advantages of Server-Side Rendering (SSR)?**
   Answer: SSR provides faster initial page loads (First Contentful Paint) because the HTML is fully generated on the server. It is also better for Search Engine Optimization (SEO) as search engine crawlers can easily index the content.

7. **Explain the concept of "Fluid Grids" in Responsive Web Design.**
   Answer: Fluid grids use relative units (like percentages) instead of fixed units (like pixels) for defining layout widths. This allows the layout to resize and adapt proportionally to the user's screen width.

PROCESS, IMPLEMENTATION & EVALUATION

1. Mapping Design to Code

The transition from design (UML diagrams, architecture) to actual code (Implementation) is a critical step. It involves translating abstract models into concrete programming constructs.

1.1 Forward Engineering

Forward Engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

- **Class Diagrams to Code**:
  - Classes become `class` definitions.
  - Attributes become member variables.
  - Operations become methods.
  - Relationships become references or collections.

Example Mapping:

UML Design:

```
classDiagram
    class User {
        -String name
        +login()
    }
```

Java Code:

```
public class User {
    private String name;

    public void login() {
        // Implementation logic
    }
}
```

1.2 Reverse Engineering

Reverse Engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.

- **Goal**: To understand legacy code, generate documentation, or recover lost design.
- **Tooling**: Many IDEs can generate UML diagrams from existing source code.

---

2. Agile Software Engineering

Agile is not just a process; it's a philosophy that values individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan.

2.1 The Scrum Framework

Scrum is the most popular Agile framework. It breaks work into short, time-boxed iterations called **Sprints** (usually 2-4 weeks).

Key Roles:

1. **Product Owner**: Represents the stakeholders and the voice of the customer. Manages the Product Backlog.
2. **Scrum Master**: Facilitates the process, removes impediments, and ensures the team follows Scrum values.
3. **Development Team**: Self-organizing group that does the actual work (coding, testing, designing).

Key Artifacts:

- **Product Backlog**: An ordered list of everything that is known to be needed in the product.
- **Sprint Backlog**: The set of items selected for the Sprint, plus a plan for delivering them.
- **Increment**: The sum of all the Product Backlog items completed during a Sprint.

Key Events:

- **Sprint Planning**: Deciding what to do in the upcoming Sprint.
- **Daily Scrum**: A 15-minute stand-up meeting to synchronize activities.
- **Sprint Review**: Inspecting the outcome of the Sprint and adapting future adaptations.
- **Sprint Retrospective**: Reflecting on the past Sprint to improve the process.

2.2 Extreme Programming (XP)

XP focuses on technical excellence and customer satisfaction. It introduces practices like:

- **Pair Programming**: Two developers work at one workstation.
- **Test-Driven Development (TDD)**: Writing tests *before* writing the code.

- **Continuous Integration**: Merging code changes frequently (multiple times a day).

---

3. Software Reuse

Software Reuse is the use of existing software, or software knowledge, to build new software. It is a key strategy to reduce cost and development time while increasing quality.

3.1 Levels of Reuse

1. **Component Reuse**: Reusing pre-built binaries or libraries (e.g., using a DatePicker component in UI).
2. **Framework Reuse**: Reusing a skeleton architecture (e.g., using Spring Boot or React).
3. **Pattern Reuse**: Reusing design patterns (e.g., Singleton, Observer).
4. **Service Reuse**: Reusing external APIs (e.g., Google Maps API, Stripe for payments).

3.2 Benefits and Challenges

Benefits	Challenges
Increased Reliability: Reused components are already tested.	**Integration Costs**: Making different components work together.
Reduced Risk: Less uncertainty in development.	**Lack of Control**: Dependency on third-party maintenance.
Faster Time-to-Market: Less code to write from scratch.	**"Not Invented Here" Syndrome**: Resistance from developers.

---

4. Architecture Documentation

Documenting architecture is essential for communicating the design to stakeholders (developers, testers, managers).

4.1 The 4+1 View Model (Kruchten)

This is a standard approach to describing software architecture using five concurrent views:

1. **Logical View**: Describes the functionality (Class diagrams). *Audience: End Users.*
2. **Process View**: Describes concurrency and synchronization (Activity diagrams). *Audience: Integrators.*
3. **Development View**: Describes the software's static organization (Package diagrams). *Audience: Programmers.*
4. **Physical View**: Describes the mapping of software to hardware (Deployment diagrams). *Audience: System Engineers.*
5. **Scenarios (+1)**: Use cases that illustrate and validate the other four views. *Audience: All.*

---

5. Architectural Evaluation

How do we know if an architecture is "good"? We evaluate it against the required quality attributes.

5.1 ATAM (Architecture Trade-off Analysis Method)

ATAM is a structured method for evaluating software architectures.

- **Goal**: To understand the consequences of architectural decisions with respect to quality attribute requirements.
- **Process**:
  1. Present the architecture.
  2. Identify business drivers (goals).
  3. Generate quality attribute utility tree (prioritize requirements).
  4. Analyze architectural approaches.
  5. Identify **Risks** (potential problems), **Sensitivity Points** (decisions that greatly affect quality), and **Trade-offs** (decisions affecting multiple qualities oppositely).

5.2 CBAM (Cost Benefit Analysis Method)

CBAM extends ATAM by adding the economic dimension. It helps answer: "How should we invest our resources to maximize the system's value?"

- It calculates the ROI (Return on Investment) for different architectural strategies.

---

6. Practice Questions

Multiple Choice Questions

Q1. What is the primary goal of Forward Engineering?

- A) To understand legacy code
- B) To generate documentation from code
- C) To move from high-level design to physical implementation
- D) To test the software

Answer: C

Q2. In Scrum, who is responsible for managing the Product Backlog?

- A) Scrum Master
- B) Development Team
- C) Product Owner
- D) Project Manager

Answer: C

Q3. Which XP practice involves writing tests before writing code?

- A) Pair Programming
- B) Continuous Integration
- C) Refactoring
- D) Test-Driven Development (TDD)

Answer: D

Q4. What does the "Physical View" in the 4+1 View Model describe?

- A) The logical functionality
- B) The mapping of software to hardware
- C) The concurrency and processes
- D) The organization of source code

Answer: B

Q5. What is the main purpose of ATAM?

- A) To calculate the cost of the project
- B) To evaluate architectural decisions against quality attributes
- C) To generate code from models
- D) To manage the Scrum team

Answer: B

Q6. Which of the following is NOT a Scrum Artifact?

- A) Product Backlog
- B) Sprint Backlog
- C) Gantt Chart
- D) Increment

Answer: C

Q7. What is the main goal of "Reverse Engineering"?

- A) To create new software from scratch
- B) To analyze existing software to understand its design
- C) To delete old code
- D) To test the software

Answer: B

Q8. In the 4+1 View Model, which view is intended for System Engineers?

- A) Logical View
- B) Process View
- C) Physical View
- D) Development View

Answer: C

Q9. Which CBAM metric helps in making architectural decisions?

- A) Lines of Code
- B) Return on Investment (ROI)
- C) Number of Classes
- D) CPU Usage

Answer: B

Q10. "Pair Programming" is a key practice of:

- A) Waterfall
- B) Extreme Programming (XP)
- C) Spiral Model
- D) V-Model

Answer: B

Short Answer Questions

1. **What is the difference between Forward Engineering and Reverse Engineering?**
   Answer: Forward Engineering is the traditional process of creating code from design models. Reverse Engineering is the opposite process of analyzing existing code to create high-level design models or documentation.
2. **List the three key roles in Scrum.**
   Answer: Product Owner, Scrum Master, and Development Team.
3. **Explain the concept of "Sensitivity Point" in ATAM.**
   Answer: A Sensitivity Point is a property of one or more components (and/or component relationships) that is critical for achieving a particular quality attribute response. For example, the level of encryption is a sensitivity point for security (but might negatively affect performance).
4. **What are the benefits of Software Reuse?**
   Answer: It increases reliability (using tested code), reduces development risk, and speeds up time-to-market by reducing the amount of new code that needs to be written.
5. **Describe the Logical View in the 4+1 View Model.**
   Answer: The Logical View supports the functional requirements of the systemâ€”what the system

should provide in terms of services to its users. It is often represented using Class Diagrams.

6. **What is the purpose of the "Sprint Retrospective" in Scrum?**
   Answer: The Sprint Retrospective is a meeting held at the end of each Sprint where the team reflects on the past Sprint to identify what went well, what didn't, and how to improve their processes and collaboration in the next Sprint.

7. **Define "Continuous Integration" (CI).**
   Answer: Continuous Integration is a development practice where developers merge their code changes into a central repository frequently (often multiple times a day), after which automated builds and tests are run to detect integration errors as quickly as possible.

REFERENCES & ABOUT THE AUTHOR

References

1. **Pressman, R. S.** (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill Education.
2. **Sommerville, I.** (2015). *Software Engineering* (10th ed.). Pearson.
3. **Bass, L., Clements, P., & Kazman, R.** (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional.
4. **Gamma, E., Helm, R., Johnson, R., & Vlissides, J.** (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
5. **Martin, R. C.** (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.
6. **Fowler, M.** (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
7. **Larman, C.** (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall.

About the Author

Gohar Rehman is a dedicated Software Engineering student at **Government Graduate College Samnabad, Faisalabad**. With a passion for building robust and scalable software systems, Gohar has compiled these notes to assist fellow students and professionals in mastering the complexities of Software Architecture and Design.

These notes are a result of rigorous study and synthesis of core software engineering principles, aiming to provide a clear, concise, and comprehensive resource for academic and professional growth.

Generated & Compiled for Educational Purposes

© 2026 Gohar Rehman. All Rights Reserved.